B.1 `doctest`

Adapted from https://docs.python.org/3.8/library/doctest.html.

The `doctest` module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown.

Here's a simple standalone example:

```
def is_even(value: int) -> bool:
    """ Return whether value is divisible by 2.

    >>> is_even(2)
    True
    >>> is_even(17)
    False
    """
    return value % 2 == 0
```

The simplest way to start using `doctest` is to end each module with:

```
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

`doctest` then examines docstrings in the module.

Running the module as a script causes the examples in the docstrings to get executed and verified.

This won't display anything unless an example fails, in which case the failing example(s) and the cause(s) of the failure(s) are printed, and the final line of output is `***Test Failed*** N failures.`, where `N` is the number of examples that failed.

You can force verbose mode by passing `verbose=True` to `testmod()`. In this case, a detailed report of all examples tried is printed to standard output, along with assorted summaries at the end.

How it works

This section examines in detail how doctest works: which docstrings it looks at, how it finds interactive examples, and how it handles exceptions. This is the information that you need to know to write doctest examples; for information about actually running doctest on these examples, see the following sections.

Which docstrings are examined?

The module docstring, and all function, class and method docstrings are searched. Objects imported into the module are not searched.

How are docstring examples recognized?

The module docstring, and all function, class and method docstrings are searched. Objects imported into the module are not searched.

In most cases a copy-and-paste of an interactive console session works fine, but doctest isn't trying to do an exact emulation of any specific Python shell.

```
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print("yes")
... else:
...     print("no")
...     print("NO")
...     print("NO!!!")
...
no
NO
NO!!!
>>>
```

Any expected output must immediately follow the final `'>>> '` or `'... '` line containing the code, and the expected output (if any) extends to the next `'>>> '` or all-whitespace line.

Notes:

- Expected output cannot contain an all-whitespace line, since such a line is taken to signal the end of expected output. If expected output does contain a blank line, put `<BLANKLINE>` in your doctest example each place a blank line is expected.
- This is an incorrect example because the prompt characters (i.e., >>>) are missing:

```
    """
    is_even(2)
    True
    """
```

- This is an incorrect example because there is no space between the `>>>` and the function call:

```
    """
    >>>is_even(2)
    True
    """
```

- This is an incorrect example because the result of the function call (`True`) is not included:

```
    """
    >>> is_even(2)
    """
```

- This is an incorrect example because the result of the function call (`True`) is indented:

```
    """
    >>> is_even(2)
        True
    """
```

What about exceptions?

The expected output for an exception must start with a traceback header, which may be either of the following two lines, indented the same as the first line of the example:

```
Traceback (most recent call last):
Traceback (innermost last):
```

The traceback header is followed by an optional traceback stack, whose contents are ignored by doctest. The traceback stack is typically omitted, or copied verbatim from an interactive session.

The traceback stack is followed by the most interesting part: the line(s) containing the exception type and detail. This is usually the last line of a traceback, but can extend across multiple lines if the exception has a multi-line detail:

```
"""
>>> 1 + 'hi'
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
"""
```

Best practice is to omit the traceback stack, unless it adds significant documentation value to the example. So the last example is probably better as:

```
"""
>>> 1 + 'hi'
Traceback (most recent call last):
TypeError: unsupported operand type(s) for +: 'int' and 'str'
"""
```

Warnings

`doctest` is serious about requiring exact matches in expected output. If even a single character doesn't match, the test fails. This will probably surprise you a few times, as you learn exactly what Python does and doesn't guarantee about output. For example, when printing a set, Python doesn't guarantee that the element is printed in any particular order, so a test like

```
>>> foo()
{'Hermione', 'Harry'}
```

is vulnerable! One workaround is to do

```
>>> foo() == {'Hermione', 'Harry'}
True
```

instead. Another is to do

```
>>> d = sorted(foo())
>>> d
['Harry', 'Hermione']
```

Soapbox

As mentioned in the introduction, `doctest` has grown to have three primary uses:

1. Checking examples in docstrings.
2. Regression testing.
3. Executable documentation / literate testing.

These uses have different requirements, and it is important to distinguish them. In particular, filling your docstrings with obscure test cases makes for bad documentation.

When writing a docstring, choose docstring examples with care. There's an art to this that needs to be learned—it may not be natural at first. Examples should add genuine value to the documentation. A good example can often be worth many words. If done with care, the examples will be invaluable for your users, and will pay back the time it takes to collect them many times over as the years go by and things change. We're still amazed at how often one of our `doctest` examples stops working after a "harmless" change.